Lecture 7
Introduction to OpenCL
Part III

John Cavazos
Dept of Computer & Information Sciences
University of Delaware

www.cis.udel.edu/~cavazos/cisc879

CISC 879 : Advanced Parallel Programming
Recap of Previous Lectures

- Get Platforms
- Get Devices
- Create Context on Host
  - Abstract container
- Create Command Queue
  - Relationship between context and device
- Create Memory Objects in Context
Recap of Previous Lectures

- Create Program object
- Build Program
  - Compiles and links program
- Create Kernel object
  - Contained in program object
Runtime Compilation

- High overhead for compiling programs and creating kernels
  - Each operation only has to be performed once (at the beginning of the program)
    - Kernel objects can be reused by setting different arguments

```
Read source code into an array

clCreateProgramWithSource

clBuildProgram

clCreateKernel
```
**Setting Kernel Arguments**

```c
cl_int clSetKernelArg(cl_kernel kernel,
cl_uint arg_index,
size_t arg_size,
const void *arg_value)
```

- Each call must specify:
  
  - Index of argument as it appears in the function signature, the size, and pointer to the data

- Examples:
  
  - `clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_iImage);`
  
  - `clSetKernelArg(kernel, 1, sizeof(int), (void*)&a);`

**CISC 879 : Advanced Parallel Programming**
Kernel Arguments

- Memory objects and individual data values can be set as kernel arguments

Data (e.g. images) are set as kernel arguments
Thread Structure

- Parallel programs are written so each thread computes one part of problem
  - Vector addition: add corresponding elements from two arrays, each thread performs one addition
Consider vector addition of 16 elements

- 2 input buffers (A, B) and 1 output buffer (C) are required

Vector Addition:

\[
\begin{align*}
A & = \\
+ & \\
B & = \\
\rightarrow & \\
C & = 
\end{align*}
\]
Thread Structure

- Create thread structure to match problem
  - 1-dimensional problem in this case

Thread structure:

Vector Addition:

\[ \begin{align*}
A + B &= C
\end{align*} \]
Thread Structure

- Each thread responsible for adding indices corresponding to its ID

Vector Addition:

\[ A + B = C \]
Thread Structure

- OpenCL’s thread structure scalable
- Each kernel is called **work-item**
- Work-items organized as work-groups
  - Work-groups independent from one-another (scalability comes from this)
- Index space defines hierarchy of work-groups and work-items
Thread Structure

CISC 879 : Advanced Parallel Programming
Thread Structure

- API calls allow threads to identify themselves and their data
- Threads can determine their global ID in each dimension
  - get_global_id(dim)
  - get_global_size(dim)
Thread Structure

- They can determine their work-group ID and ID within the workgroup
  - `get_group_id(dim)`
  - `get_num_groups(dim)`
  - `get_local_id(dim)`
  - `get_local_size(dim)`
  - `get_global_id(0) = column, get_global_id(1) = row`
  - `get_num_groups(0) * get_local_size(0) == get_global_size(0)`
Memory Model

Accessible by all work-items

CISC 879 : Advanced Parallel Programming
Memory Model

CISC 879: Advanced Parallel Programming

- Read-only, global
Memory Model

Local to a work-group
Memory Model

Private to a work-item
Memory Model

- Memory management is explicit
- Work-groups are assigned to execute on compute-units
  - No guaranteed communication/coherency between different work-groups
• One kernel instance created for each thread

• Kernels:
  • Must begin with keyword __kernel
  • Must have return type void
  • Must declare the address space of each argument that is a memory object (next slide)
  • Use API calls (such as get_global_id()) to determine which data a thread will work on
Address Space Identifiers

- __global
  - memory allocated from global address space
- __constant
  - a special type of read-only memory
- __local
  - memory shared by a work-group
- __private
  - private per work-item memory
Executing the Kernel

- Need to set dimensions of index space, and (optionally) of work-group sizes
- Kernels execute asynchronously from host
  - `clEnqueueNDRangeKernel` adds to queue
    - Doesn’t guarantee that it will start executing
Executing the Kernel

- A thread structure defined by index-space created
  - Each thread executes the same kernel on different data

An index space of threads created (dimensions match the data)
Executing the Kernel

- A thread structure defined by index-space created
  - Each thread executes the same kernel on different data
Executing the Kernel

cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
cl_kernel kernel,
cl_uint work_dim,
const size_t *global_work_offset,
const size_t *global_work_size,
const size_t *local_work_size,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)

- Tells device associated with command queue to begin executing the specified kernel
- Global (index space) must be specified and the local (work-group) sizes are optionally specified
- List of events can be used to specify prerequisite operations that must be complete before executing
Copying Data Back

- Last step is to copy data back from device to host
- Similar call as writing a buffer to a device

```c
cl_int clEnqueueReadBuffer(cl_command_queue command_queue,
                           cl_mem buffer,
                           cl_bool blocking_read,
                           size_t offset,
                           size_t cb,
                           void *ptr,
                           cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list,
                           cl_event *event)
```
Copying Data Back

- The output data is read from the device back to the host.
Most OpenCL resources/objects are pointers
  - should be freed after being used
There is a clRelease{Resource} command for most OpenCL types
  - For example:
    - clReleaseProgram()
    - clReleaseMemObject()
Error Checking

- OpenCL commands return error codes as negative integer values
  - Return value of 0 indicates CL_SUCCESS
  - Negative values indicates an error
    - cl.h defines meaning of each return value

Note: Errors sometimes reported asynchronously

CL_DEVICE_NOT_FOUND -1
CL_DEVICE_NOT_AVAILABLE -2
CL_COMPILER_NOT_AVAILABLE -3
CL_MEM_OBJECT_ALLOCATION_FAILURE -4
CL_OUT_OF_RESOURCES -5
Big Picture

OpenCL

Context

Programs

Kernels

Memory Objects

Command Queues

Compile code

Create data & arguments

Send to execution

_CISC 879 : Advanced Parallel Programming_
Programming Model

- Data parallel
  - One-to-one mapping between work-items and elements in a memory object
  - Work-groups can be defined explicitly (like CUDA) or implicitly (specify the number of work-items and OpenCL creates the work-groups)

- Task parallel
  - Kernel is executed independent of an index space
  - Other ways to express parallelism: enqueueing multiple tasks, using device-specific vector types, etc.

- Synchronization
  - Possible between items in a work-group
  - Possible between commands in a context command queue
Summary

• OpenCL provides interface for interaction of hosts with accelerator devices

• Context is created that contains all of information and data required to execute an OpenCL program
Summary

- Memory objects are created that can be moved on and off devices
- Command queues allow the host to request operations to be performed by the device
- Programs and kernels contain the code that devices need to execute